REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	10/1/98	Final Technical Report 5/91-9/94

4. TITLE AND SUBTITLE
A Directory-Based Scalable General-Purpose Shared-Memory Multiprocessor

5. FUNDING NUMBERS
N00039-91-C-0138

6. AUTHOR(S)
John L. Hennessy

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Stanford University
Terman Engineering Bldg, rm. 214
Stanford, CA 94305

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 N. Fairfax
Arlington, VA 22203

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This research focused on the design and development of scalable shared-memory machines, in particular, those using directory-based cache coherence. This research led to the design and fabrication of the machine - the Stanford DASH machine.

14. SUBJECT TERMS

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

A Directory-Based Scalable General-Purpose Shared-Memory Multiprocessor

Stanford University

Computer Systems Laboratory

Final Technical Report

Defense Advanced Research Projects Agency

May 1991 – September 1994

Contract Number: N00039-91-C-0138

AO #7540

Principal Investigator

John L. Hennessy

(jlh@vsop.stanford.edu)

Co-principal Investigator

Mark A. Horowitz

(horowitz@chroma.stanford.edu)

Overview

This research project focused on the design and development of scalable shared-memory machines; in particular, those using directory-based cache coherence. The basic mechanisms of directory-based cache coherence were developed in an earlier research program, leading to the design and initial bring-up of the Stanford DASH prototype, the first operational multiprocessor to support scalable cache coherence. This project involved scaling the DASH machine to larger processor counts (64 vs. 16), a through evaluation of the design, completion of new multiprocessor software systems, and the initial design of a second-generation, more flexible machine, called FLASH. In addition, several basic studies of parallel architecture, multiprocessor technology, application parallelization, and related problems were undertaken. This report summarizes the technical accomplishments of this project, organizing the results by project or area. A complete list of publications appears at the end.

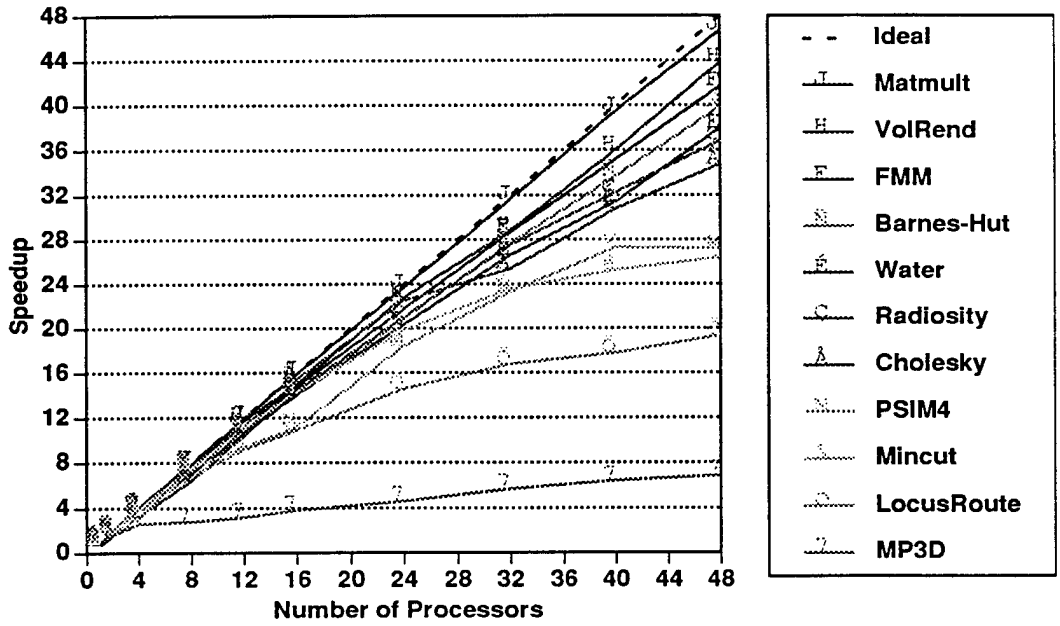
The DASH Project

We began this research contract with a barely operating 16-processor DASH prototype. During the first 24 months of the project, we brought the machine up to 64 processors (though reliability problems in the routing chip motivated us to use 48 processors as the usual operating mode), completed the operating system support, ported a range of applications, developed several novel performance monitoring and analysis tools, and extensively measured and analyzed the machine. The design of DASH was detailed extensively in a book by Lenoski and Weber, and this book, together with other publications, has been extensively used by industry as a handbook for building scalable cache-coherent multiprocessors. Several companies (HP/Convex, Silicon Graphics, Sequent, HaL, and Data General) have built products partially or totally based on the ideas pioneered in DASH. In addition, a number of other companies (IBM, Sun, Compaq) have been investigating the distributed directory approach that the DASH project invented.

The DASH Prototype

We completed the DASH prototype in 1992 and began extensive measurements. The primary DASH machine consisted of 48 processors, and was used for application performance tuning and evaluation of the DASH hardware features. We also used two eight processor machines, used for initial application development and debug, and operating system development. We had several applications running on DASH. Most of the applications got good speedups on 48 processors, even without a large effort spent in performance-tuning.

We completed several studies that compared DASH to the KSR-1 machine, the only other scalable cache-coherent machine available at the time. This work extended our previous simulation-based comparison of the COMA and CC-NUMA architectural styles by using full-scale applications on real hardware. We ran full-scale problems on real COMA and CC-NUMA machines (KSR-1 and DASH) which have similarly sized caches. We used up to 48 processors on the two machines (2 rings on KSR-1). For small problems, we found DASH speedups to be invariably much better than those of KSR-1, since small problems have high communication-to-computation ratios and greater false-sharing (especially with the long cache lines on KSR-1), but not many capacity misses. For applications with low



miss rates and relatively small working sets (N-body computations, blocked dense linear algebra, etc.), we found both machines to yield about equally good speedups, DASH being a little bit better. And for the applications that had significant capacity misses, we found that (i) it was reasonably easy to manage data distribution well on DASH, so that DASH yielded equivalent or better speedups, and (ii) the techniques used to do this helped performance on KSR-1 substantially as well, since they reduced conflict misses and false sharing. KSR-1 did better on applications that needed high-bandwidth communication of large chunks of contiguous data (such as FFT), but this was due to peak communication bandwidth and not due to its COMA nature. The only example in which the COMA nature of KSR-1 was a clear advantage was the dynamically scheduled panel-oriented Cholesky factorization kernel from SPLASH.

The DASH prototype was kept operational as an important tool for studying multiprocessor designs, and was decommissioned in 1997, approximately six years after the first 16 processors began working and five years after the full machine became operational.

The FLASH Multiprocessor

In mid-1993, we began the design of the Stanford FLASH (Flexible Architecture for SHared Memory) Multiprocessor. A key goal for the FLASH architecture is to support both coherent shared memory and high performance message passing with minimal hardware overhead in a scalable manner. Each node in FLASH is identical, containing a high-performance off-the-shelf microprocessor with its caches, a portion of the machine's distributed main memory, and the MAGIC node controller chip. The MAGIC chip forms the heart of the node, integrating the memory controller, I/O controller, network interface, and a programmable protocol processor. This integration allows for low hardware overhead, while supporting both cache-coherence and message-passing protocols in a scalable and cohesive fashion.

The Core of the FLASH Design–The Magic Chip

To offer both flexibility and high performance, the MAGIC controller has several unique features. First, MAGIC includes a programmable protocol processor for flexibility. Second, MAGIC's central location within the node ensures that it sees all processor, network, and I/O transactions, allowing it to control all node resources and support a variety of protocols. Third, to avoid limiting the node design to any specific protocol and to accommodate protocols with varying memory requirements, the node contains no dedicated protocol storage; instead, both the protocol code and protocol data reside in a reserved portion of the node's main memory. However, to provide high-speed access to frequently-used protocol code and data, MAGIC contains on-chip instruction and data caches (due to a change in the fabrication source, the data cache was subsequently moved off-chip). Finally, MAGIC separates data movement logic from protocol state manipulation logic. The hardwired data movement logic achieves low latency and high bandwidth by supporting highly-pipelined data transfers without extra copying within the chip. The protocol processor employs a hardware dispatch table to help service requests quickly, and a coarse-level pipeline to reduce protocol processor occupancy. This separation and specialization of data transfer and control logic ensures that MAGIC does not become a latency or bandwidth bottleneck.

At the core of the MAGIC chip is the protocol processor, which is used to perform all computation necessary to provide coherent-cache and message-passing primitives. We evaluated the performance gains due to various features in the instruction set of the protocol processor. Weighing the gains of each feature against its implementation cost has allowed us to define a base architecture for the processor that is well suited for efficient execution of the required primitives. The protocol processor itself is a dual-issue superscalar RISC CPU with instruction set extensions for rapidly manipulating bit fields, a common task in maintaining directory state. The basic RISC core is surrounded by specialized hardware units that queue incoming messages from the processor and the network, quickly dispatch the protocol processor to the code for the primitive indicated by those messages (e.g., a memory read), and send reply messages to the processor and network under the direction of the protocol processor. Through the pipelining of these units and the optimization of the protocol processor instruction set, this architecture achieves the high throughput requirements for state manipulation that accompany the high-bandwidth data transfer rates of

next-generation memory systems.

Using the protocol processor, we have also developed a high performance message passing implementation cleanly integrated with cache coherence. Simulation results indicate that we can achieve comparable performance to message passing machines with dedicated hardware support. Furthermore, our implementation offers increased integration of machine attributes such as protection, virtual memory, and multiprogramming that has been missing in other implementations.

As of the close of this project, we completed most of the Verilog description of Magic (with I/O being the major missing task). Although we have done extensive simulation of Magic, validation of the Magic Verilog, both for functionality and performance remains to be done. In addition, we have not yet explored the task of synthesis from the Verilog description.

FLASH Message Passing

The advantages of using message passing over shared memory for certain types of communication and synchronization operations have provided an incentive to integrate both communication modes within a single architecture. One of the main goals of the FLASH architecture is to achieve this integration while maintaining a simple and efficient design.We have developed hardware and software mechanisms to support various message passing protocols in FLASH. We found we can achieve low overhead message passing by delegating protocol functionality to the programmable node controllers in FLASH and by providing direct user-level access to this messaging subsystem. In contrast to most earlier work, we provide an integrated solution that handles the interaction of the messaging protocols with virtual memory, protected multiprogramming, and cache coherence. Our preliminary performance studies, based on detailed simulations of several message passing primitives, indicate that this system can sustain message transfers at a rate of several hundred megabytes per second, effectively utilizing projected network bandwidths for next generation multiprocessors.

Integration of Block Data Transfer in Cache-Coherent Multiprocessors

Many of the shared-memory multiprocessors developed, such as the Stanford FLASH, MIT Alewife, and Wisconsin Typhoon multiprocessors, provide architectural support for block data transfer. We have examined the potential performance benefits from using block transfer in a cache-coherent shared address space multiprocessor. A set of ambitious hardware mechanisms was used to study the performance gains obtained in five important computations which appeared to be good candidates for using block transfer: (i) FFT, (ii) LU Decomposition, (iii) Sparse Cholesky Factorization, (iv) SPLASH Ocean Simulation, and (v) Radix Sort. Our conclusion is that the benefits of block transfer are not substantial for hardware cache-coherent multiprocessors. The main reasons for this are (i) the relatively modest fraction of time applications spend in communication amenable to block transfer, (ii) the difficulty of finding enough independent computation to overlap with the communication latency that remains even after block transfer, and (iii) that long cache lines often capture many of the benefits of block transfer.

In the cases where block transfer improved performance, prefetching provided comparable, if not superior, performance benefits compared to block transfer. We also examined the

impact of varying important communication parameters and increasing processor speed on the effectiveness of block transfer. One important result we uncovered is that the advantages of block transfer are greatest in shared memory implementations which have high overhead for initiating communication, but that also provide high bandwidth for communication, such as on message passing machines and networks of workstations. Finally, we examined useful features that a block transfer facility should support for real applications.

General Multiprocessor Architecture Studies

Memory Models

The memory consistency model of a shared-memory system determines the order in which memory accesses can be executed by the system, and greatly affects the implementation and performance of the system. To aid system designers, memory models either directly specify, or are accompanied by, a set of low-level system conditions that can be *translated* into a correct implementation. These sufficient conditions play a key role in helping the designer determine the architecture and compiler optimizations that may be safely exploited under a specific model. Therefore, these conditions should obey three important properties. First, they should be unambiguous. Second, they should be feasibly aggressive; i.e., they should not prohibit practical optimizations that do not violate the semantics of the model. Third, it should be relatively straightforward to convert the conditions into efficient implementations, and conversely, to verify if an implementation obeys the conditions. Most previous approaches in specifying system requirements for a model are lacking in at least one of the above aspects.

Our work presents a methodology for specifying the system conditions for a memory model that satisfies the above goals. A key attribute of our methodology is the exclusion of ordering constraints among memory operations to different locations by observing that such constraints are unnecessary for maintaining the semantics of a model. To demonstrate the flexibility of our approach, we specify the conditions for several proposed memory models within this framework. Compared to the original specification for each model, the new specification allows more optimizations without violating the original semantics and, in many cases, is more precise.

The Benefits of Clustering in Cache-Coherent Multiprocessors

Clustering processors together at a level of the memory hierarchy in shared address space multiprocessors appears to be an attractive technique from several standpoints: resources are shared, packaging technologies are exploited, and processors within a cluster can share data more effectively. We have investigated the performance benefits that can be obtained by clustering on a range of important scientific and engineering applications. We found that in general clustering is not very effective in reducing inherent communication to computation ratios. Clustering is more useful in reducing working set requirements in unstructured applications, and can improve performance substantially when small first level caches are clustered in these cases. This suggests that clustering at the first level cache might be useful in highly-integrated, relatively fine-grained environments. For less integrated machines such as current distributed shared memory multiprocessors, our results suggest that at least in the absence of artifacts such as cache mapping collisions, clustering is not very useful in improving application performance, and the decision about whether or not to cluster should be made on the basis of engineering and packaging constraints.

Advantages and Challenges in COMA Architectures

We are investigating the design issues involved in building a COMA-Flat multiprocessor (our own extension to the standard COMA protocols). While traditional COMA architec-

tures such as the Kendall Square Research KSR1 and the Swedish Institute of Computer Science DDM utilize dynamic binding of data objects and directory information to physical locations to improve performance, this necessitates the use of relatively slow directory hierarchies. THE COMA-Flat protocol supports the dynamic binding of data objects to physical locations without the need of such a hierarchy, permitting the use of a high speed general interconnection topology such as a mesh.

We have investigated the problem of memory block replacements in COMA architectures. When a memory block is requested by a processor and the local memory does not have space to hold it, an existing memory block is chosen for replacement. If this block represents the last copy of a data object, it must be sent to another memory for storage. When the frequency of this occurrence is high, system performance may become degraded. We analyzed this replacement traffic and developed a mathematical model which predicts their frequency based upon the memory organization and size. We found that when the application data set size is large compared to the total memory size, the number of replacements generated is extremely high. We also find the number of replacements is mostly insensitive to the block size, but quite sensitive to the associativity of the memory. Even small degrees of associativity will significantly reduce the number of replacements generated in situations of high memory usage. For example, we find that when the ratio of the application data set size to the memory size is 85%, a 4-way associative memory generates 1.09 replacements per miss. However, a direct mapped memory will generate 5.67 replacements per miss. These studies illustrate that a competitive COMA implementation will have to pay close attention to its potential disadvantages to ensure that it delivers good performance.

Parallel Software

SUIF Parallel Compiler System

One major goal of this work was to create a suitable platform for building parallel compiler systems. The initial version of SUIF was released to over 20 research institutions for this purpose. Our SUIF system is based on an "open-system" design, where various compiler passes are integrated together through a common interface. We expect this design to be more usable by outside compiler groups than previously released systems, because any of the phases in the compiler can be readily replaced by other modules with the same interface. This design enables our group to focus only on the core interface, a task that is much more manageable than maintaining the entire system. This approach of developing a standard interface appears to be a practical solution to the outstanding problem of how to provide an infrastructure for the compiler research community.

Interprocedural Optimization for Parallelization

We have developed an interprocedural optimization system, the FIAT system, a framework to facilitate rapid prototyping of interprocedural systems. FIAT, which was developed in collaboration with researchers at Rice University, used to drive interprocedural optimization in both the SUIF compiler and the D Programming Tools at Rice. Our research focused on using FIAT to build a comprehensive suite of analyses for parallelization, in order to measure how effective an aggressive interprocedural system is at locating parallelism across procedure boundaries.

One criterion for evaluating an interprocedural parallelization system is its ability to parallelize loops containing procedure calls. Such loops are typically not parallelized by existing compilers because, without interprocedural analysis, the compiler does not know how the called procedure will affect the safety of parallelization. Previous studies of interprocedural systems for parallelization have been fairly successful at parallelizing loops with calls in linear algebra libraries, which are cleanly written and exhibit fairly regular array access patterns. However, these systems have not been effective at parallelizing full application programs such as the Perfect benchmark suite. One reason for their limited success is that previous interprocedural systems did not provide interprocedural counterparts for all the analysis techniques currently used in the intraprocedural setting.

Our system provides the same quality of interprocedural analysis as would be available in a state-of-the-art intraprocedural compiler. Because many of these analyses must precisely describe program behavior, this project required extending FIAT to support flow-sensitive interprocedural analysis in addition to its existing flow-insensitive analysis; flow-sensitive analysis examines control flow paths within procedures during interprocedural propagation to determine what facts are true along all control flow paths. Efficient approaches to flow-sensitive analysis are currently an open area of research. Flow-sensitive analysis can be computationally infeasible using iterative data-flow analysis techniques, which are commonly used in the intraprocedural setting. We discovered that the flow-sensitive analysis data-flow problems we were solving were amenable to an interval style of analysis, thus providing a practical solution to obtaining precise analysis.

We measured FIAT's ability to parallelize loops containing procedure calls. In comparisons with earlier interprocedural systems, our system is significantly more effective at parallelizing loops containing procedure calls in the Perfect and SPEC benchmark programs. The difference in our results stem from the requirement, in many of these loops, for a number of distinct interprocedural analysis techniques working together.

Prefetching by Compilers

Software-controlled data prefetching is a promising technique for improving the performance of the memory subsystem to match today's high-performance processors. While prefetching is useful in hiding the latency, issuing prefetches incurs an instruction overhead and can increase the load on the memory subsystem. As a result, care must be taken to ensure that such overheads do not exceed the benefits.

We have developed a compiler algorithm to insert prefetch instructions into code that operates on dense matrices. Our algorithm identifies those references that are likely to be cache misses, and issues prefetches only for them. We have implemented our algorithm in the SUIF optimizing compiler. By generating fully functional code, we have been able to measure not only the improvements in cache miss rates, but also the overall performance of a simulated system. We show that our algorithm significantly improves the execution speed of our benchmark programs–some of the programs improve by as much as a factor of two. When compared to an algorithm that indiscriminately prefetches all array accesses, our algorithm can eliminate many of the unnecessary prefetches without any significant decrease in the coverage of the cache misses.

Compiler Optimizations for Shared Address Space Machines

At first glance, it seems much easier to compile to shared address space machines (such as the DASH and KSR-1) as compared to distributed address space machines (such as the Paragon and SP-2), since the programmer need not explicitly manage communication for non-local data. However, while it is relatively simple to get a program to run correctly, it is non-trivial to get scalable performance. As scalable parallel machines tend to have non-uniform memory access times, we find the compiler can significantly benefit from the analyses and optimizations performed to minimize communication on distributed memory machines.

We investigated three optimization techniques: parallelism and locality analysis, communication and synchronization analysis, and changing data layouts. During parallelism and locality analysis, the compiler first optimizes parallelism at the loop level by reordering the computation to discover the largest granularity of parallelism using unimodular code transformations (e.g. loop interchange, skewing and reversal). Global analysis then examines all loop nests together to determine an overall mapping of data and computation across the processors such that parallelism is maximized while minimizing communication.

During communication and synchronization analysis, the compiler uses the data mapping information to identify accesses to non-local data. This step is not required to ensure correctness as on distributed-memory machines, but is used to optimize synchronization. Many parallelized programs consist of a large number of parallel loops, each of which does not contain much computation. The resulting profusion of barriers incurs high overhead and inhibits parallelism. By taking advantage of the fact that data and computation map-

pings are calculated at compile time, we can use information on when and where communication is necessary to eliminate unnecessary barrier synchronization or replace them with efficient point-to-point synchronization.

Finally, we found that changing the data layout can greatly improve the cache performance. Real caches have long cache lines and limited set associativity. One simple way to enhance spatial locality, minimize false sharing, and reduce cache interference is to restructure the array so that the major regions of data accessed by a processor are contiguous in memory. During code generation, the compiler translates array subscripts in the original code to accesses for the new array layout calculated in the parallelism and locality phase. If the new organization has more dimensions than the original, the address calculations now include division and modulo operations. A set of special optimizations are used to eliminate most of these division and modulo operations.

Performance Debugging Tools

MTOOL, initially developed under earlier funding, was improved and put into active use. MTOOL supports not only C with the ANL macros and compiler parallelized Fortran but also our experimental parallel languages COOL and Jade. MTOOL has been successfully used to tune several of the SPLASH applications programs (PSIM4, Barnes-Hut, Radiosity, cholesky) and in conjunction with a parallelizing compiler, has helped reduce the run-time of a significant application from Economics by a factor of 7 on an 8 processor SGI system. Several MTOOL ports were done by outside companies. For example, SUN ported a version of MTOOL to Sun SuperSparc-based multiprocessors.

We developed a new type of a performance debugging tool: MemSpy is a software tool which provides users with *detailed, data-oriented* information on an application's memory system performance. For all application data objects and procedures, MemSpy provides information on the amount of memory stall at the time they incur, and breakdowns of the causes of this memory stall time. This information has proven useful in uncovering performance bottlenecks in a number of applications. A primary focus of work on MemSpy has been to improve its execution time overhead. Although MemSpy is simulation based, proper optimization of simulator performance should allow its overhead to be competitive with tools such as MTOOL, which have traditionally opted for lower overhead by providing less detailed information on application performance. Initially, using MemSpy on an application incurred factors of roughly 20 to 50 times slowdown, depending on the specific applications memory referencing characteristics. With many optimizations, MemSpy's overhead has been brought to 7 to 20 on applications from the SPLASH benchmark set. MemSpy was released for general distribution in 1993.

COOL-An Object-Oriented, Explicitly Parallel Programming Language

Building on the concepts from the object-oriented community, we developed a new programming language called COOL (Concurrent Object Oriented Language) that uses explicit parallel constructs and monitors as basic programming tools. We implemented COOL on DASH, developed new applications in COOL including programs from the SPLASH benchmark suite, and studied various performance issues.

The COOL programming language emphasizes the integration of concurrency and syn-

chronization with data abstraction. This approach provides several benefits. First, integrating concurrency with data abstraction allows construction of concurrent data structures that have most of the complex details suitably encapsulated. Second, monitors and condition variables integrated with objects offer a flexible set of building blocks that can be used to build more complex synchronization abstractions. Synchronization operations are clearly identified through attributes and can be optimized by the compiler to reduce synchronization overhead. Finally, the object framework supports abstractions to improve the load distribution and data locality of the program, thus offering higher performance.

We developed optimizations to efficiently support synchronization through monitors, and techniques to improve data locality. Synchronization built using monitors can suffer from high implementation overheads. To address this problem, we have developed compiler optimizations that analyze monitor operations, and, based on the synchronization they express, automatically optimize their implementation. These optimizations can significantly reduce the overheads for several common instances of monitor usage.

Regarding techniques to improve data locality, we have employed knowledge of the underlying memory hierarchy to schedule computation and distribute data structures and thereby improve data locality. We have developed abstractions for the programmer to supply information about the data reference patterns of the program. This information is used by the runtime task scheduler to distribute tasks and objects so that tasks execute close (in the memory hierarchy) to the objects they reference. We evaluated the effectiveness of these techniques by applying them to applications from the SPLASH benchmark suite.

COOL was completed in 1993 and released to the research community. Several outside groups have used COOL for programming object-oriented, parallel applications.

JADE–A Parallel Programming Language

Unlike many other conventional parallel language constructs that are control-oriented, Jade is data-oriented. A Jade programmer parallelizes an application by augmenting sequential code with Jade constructs that declare the side effects of different sections of the code. The compiler and run-time system use that side effect information to execute the program in parallel while still enforcing the data dependence constraints implied by the original serial program. Jade's data-oriented approach simplifies development of large parallel applications. The programmer only expresses local data usage information about each task; the Jade implementation determines the (possibly quite complex) synchronization necessary between tasks. We have discovered that data-oriented expression of parallelism can be used to describe sophisticated concurrency patterns that have been developed with control-oriented languages. A program's concurrency structure is captured in the design of the program's data structures. Simple data structures can be used for simple concurrency patterns such as dynamic task graphs and pipelining. More complicated patterns such as nested levels of parallelism can be achieved via hierarchically structured data.

Hierarchically structured data allow the programmer to express tasks' accesses at different levels corresponding to how the tasks access the data. In this way, the programmer can express a task's accesses at the most appropriate level for that task. Hierarchies can also be useful for refining the access specification of a task which incrementally narrows down the

set of data it can potentially access. When a programmer expresses how tasks access data in this hierarchical manner, the Jade implementation can fully exploit the natural concurrency available within and among operations of hierarchically structured data. The resulting hierarchical concurrency patterns also naturally make the generation of the dynamic task graph in a Jade program more efficient.

We have used Jade to parallelize large applications, such as a simulation of the air flow in the Los Angeles basin, a 3-D graphical modeling system, and a program using finite-element analysis. Jade was also ported to the Stanford DASH multiprocessor prototype relatively easily. We released JADE in 1994 and several companies and research groups have used JADE to explore a variety of multiprocessing tasks ranging from other shared-memory multiprocessors to networks of workstations.

Parallel Application Studies

As part of the DASH project, we engaged in a variety of application studies. Several important results came of these studies. First, the development and characterization of the SPLASH benchmark set. SPLASH (Stanford ParaLlel Applications for SHared Memory) is the first publicly available set of scalable, parallel applications for shared memory multiprocessors. SPLASH is distributed via the web (including reports that characterize and document the applications). SPLASH has been widely downloaded and used by a variety of researchers working on both parallel hardware and parallel software systems.

Our second major thrust was to explore the parallelization of challenging applications. We have focused on the increasingly important n-body applications. These applications have promising uses and potentially good speed-up but present new challenges in memory locality and potentially in load balancing. Our work has demonstrated new approaches to solving these problems and clearly illustrates the advantages of shared-memory architectures. In particular, our work has focused on the two best methods for classical N-body problems–the Barnes-Hut method and the Fast Multipole Method (FMM)–as well as on a very different application of the hierarchical N-body approach: a hierarchical radiosity application for global illumination problems in computer graphics. Although obtaining effective parallel performance in these applications is complicated by their nonuniform and dynamically changing characteristics, we have designed new partitioning techniques that yield very good speedups on a 48-processor DASH. We have also studied the implications of these hierarchical N-body applications for multiprocessor architecture. Our first goal was to study the implications of scaling the applications to run on larger machines. We developed a realistic scaling methodology that takes all relevant application parameters into account, and demonstrated that it leads to different conclusions about the effectiveness and design of larger machines than the popular, naive method of considering only the impact of scaling the input data set size. Besides scaling, our other architectural conclusion from studying these applications had to do with the kind of communication abstraction that a multiprocessor should support. We found that the nonuniform and dynamically changing nature of the problem domain, together with the need for long-range communication, cause a shared address space to afford substantial advantages in both programming complexity as well as performance over an explicit message-passing communication paradigm.

We have also been looking at scalability issues for the important sparse Cholesky factor-

ization computation. We have investigated the use of a *panel* decomposition for parallel sparse factorization, where sets of contiguous matrix columns with identical non-zero structure are distributed among the processors. Through extensive performance modelling and implementations on the DASH machine, we have learned several important things about such methods. First, panel methods produce a significant performance improvement over the more traditional column methods due to improved use of the memory hierarchy (a factor of two to three performance increase is typical). However, their performance is still somewhat disappointing, since they produce speedups that are well below linear in the number of processors. Our performance modelling work has allowed us to understand the achieved performance in terms of fundamental limitations in the computation. Indeed, we have found that such methods have very severe limitations due to a lack of exposed concurrency. Concurrency is inadequate for the moderately parallel machines we considered, and it would prove to be an even more severe limitation for larger machines.

FLASH Operating System

Our goal is an operating system that makes FLASH usable in general-purpose (commercial and engineering) environments in addition to supercomputer environments. As of the end of this contract, we completed several major steps towards this goal:

1. Acquired and ported a commercial UNIX system to DASH as a base for FLASH OS development.
2. Developed a detailed measurement method (using the monitoring hardware on the DASH) and used it to characterize the effects of NUMA on the OS we ported when run under a general-purpose workload. The information gathered, about where the performance problems occur and, just as importantly, about where they do not occur, is vital to planning our development efforts for FLASH.
3. Used DASH to study the benefits and costs of several different scheduling and memory page migration policies on application performance in a general-purpose workload.
4. Developed a simulation environment which can efficiently simulate an operating system and applications, rather than just applications as previous efficient simulation methods could handle (for example, Tango).
5. Completed the first revision of the high-level architecture, and begun work on the file system and I/O architecture.

To further our understanding of how cache-coherent NUMA multiprocessors will behave under the workloads presented to general-purpose compute servers, we have undertaken two different paths of study. One path involved studying workloads running on the DASH prototype machine. The other involved building a simulation environment capable of simulating the FLASH machine running complex multiprogramming workloads.

Using the DASH machine, we studied the effects of OS scheduling and page migration policies on the performance of multiprogramming workloads. Results of this study are available in the paper entitled "Scheduling and Page Migration for Multiprocessor Compute Servers" which was presented at ASPLOS. Our experiments show that for

multiprogramming workloads consisting of sequential jobs, the UNIX (SVR3) scheduling policy does very poorly on machines such as DASH. By incorporating cluster and cache affinity along with a simple page-migration algorithm, performance can be increased by up to a factor of two. For workloads consisting of multiple parallel applications, we show that space-sharing policies that divide the processors among the applications do substantially better than time-slicing policies such as standard UNIX or gang-scheduling. Finally, we evaluated using TLB misses as a predictor of cache misses for page migration in parallel applications.

Operating System and Simulation Technology

To understand the behavior of multiprogrammed workloads on FLASH, we have developed an interface so the SimOS simulation environment can be used to drive the flashlite memory system simulator. SimOS, described in the paper entitled "Fast and Accurate Multiprocessor Simulation: The SimOS Approach", is a machine simulation environment that contains a full Unix operating system. Having the full operating system allows SimOS-based simulations to both include the OS behavior in the simulation results as well as enabling complex workloads to run in the simulation environment. The current SimOS environment is application-level binary compatible with the system software environment being built for FLASH.

Beside supporting complex, realistic, and long-running workloads, SimOS also supports the flexibility to trade-off speed and detail. Faster, less detailed simulation can be used to scan over the uninteresting, time-consuming parts of an execution while slower, more detailed levels of simulation can be used to focus on specific sections of interest. SimOS's ability to change levels of detail on-the-fly enhances its ability to study complex workloads.

By interfacing to FLASHLite (the FLASH memory system simulator), we have created an environment which allows us to switch to a level of simulation detail that closely models both the hardware, system software, and workloads that will be running on the FLASH machine. We have already used this capability to evaluate the FLASH machine on multiprogrammed workloads[8]. Even though it has come on-line only recently, this simulation environment has already given us interesting insight into the behavior of the design.

Uniprocessor Architecture Studies

Exploiting Instruction Level Parallelism

A growing perception is that dynamically-scheduled superscalar processors are the only effective way to couple instruction scheduling and speculative execution. This perception seems to be supported in the commercial world as superscalar implementations move from dynamic dependence checking (e.g. the Sun SuperSPARC) toward more complex dynamic scheduling techniques with support for speculative execution (e.g. the Motorola 88110). Yet, this hardware-intensive approach has a fundamental problem: these machines analyze only a small window of instructions at a time and use simplistic heuristics for choosing among the available instructions. Thus, they are not guaranteed to generate a good instruction schedule.

On the other hand, compiler algorithms that incorporate global scheduling techniques can effectively schedule instructions across branch boundaries. These techniques have advantages over run-time instruction scheduling because they are able to analyze a much larger portion of the program at any time, and they can use sophisticated heuristics to choose among the available instructions. These advantages allow the compiler to optimize the schedule for the critical paths of the program. While these compiler-based approaches have the benefit of much simpler issue hardware, they have been limited in their ability to use speculative execution. To augment these global scheduling algorithms, we recently proposed a general architectural mechanism called boosting that provides the compiler with an unconstrained model of speculative execution. Since then, we have constructed a complete compiler system and a working hardware model to better understand the capabilities and costs of boosting.

Our TORCH project attempts to find an integrated solution, consisting of both compiler and architectural components, to the problem of effectively exploiting ILP. Our focus is on supporting general speculative execution in an environment with sophisticated instruction scheduling. Speculative execution is needed to find any significant amount of ILP in non-numerical applications. The key idea in our approach is in defining the right architectural interface; an interface that appropriately divides the work among the hardware and software. Our boosting approach relies on simple but useful hardware mechanisms that minimally impact the cycle time of the machine. Our preliminary hardware designs indicate that the cycle time of a superscalar machine with boosting past one conditional branch is nearly identical to the cycle time of a simple (VLIW-like) superscalar machine. By making these hardware mechanisms visible to the software, a sophisticated static instruction scheduler can take full advantage of speculative execution with very little cost.

We have developed a global scheduling algorithm that is useful for machines with and without boosting. Our algorithm is applicable to a wide range of machines, from deeply-pipelined RISC processors to dynamically-scheduled superscalar machines. Our algorithm uses a trace-based approach to efficiently exploit the ILP in non-numerical applications. Like other trace-based approaches, our algorithm concentrates on those code motions that are most beneficial to improving performance. Unlike other trace-based approaches, our algorithm tempers the global movements so as to not adversely affect the performance of the off-trace schedules.

As a result of our compiler implementation and in-depth hardware design, we have been able to evaluate our ideas and study a range of cost/performance tradeoffs. For the limited superscalar machines of the early 1990s, our statically-scheduled approach achieves cycle count speedups which are comparable to those found in the aggressive dynamically-scheduled approaches.

As an additional benefit of this work, we are using the TORCH compiler scheduling system to create the compiler back-end for the Magic protocol processor.

Hardware and Software Technology for Multiprocessors

Formal Verification of Multiprocessor Protocols

Protocols are susceptible to subtle design errors which are difficult to detect and diagnose by simulation or prototyping. Our goal is to speed up the design of correct protocols by providing software for automatic formal verification. Our approach is to describe the protocol using a simple language that we have devised, TRANS. Protocols described in TRANS can be checked automatically for violations of invariant properties or deadlocks.

The TRANS language is based on iterated guarded commands, like Chandy and Misra's UNITY language, supporting data structures such as records and arrays. Simplicity in the language design was a high priority, because we want to apply a variety of verification techniques to TRANS descriptions. Every additional feature in the language would make adaptation to a new method more difficult.

We have implemented the simplest possible verification methods, so that we could explore examples in TRANS in parallel with the development of more sophisticated methods. The first methods are based on explicit enumeration of the states of the protocol. Each state is added to a table after being checked for deadlock or violation of a user-specified invariant condition. Compared with other programs using the same basic method, we put more effort into minimizing the sizes of individual states, so we can handle somewhat larger problems in a given amount of memory.

We pursued three subprojects; debugging the verification system, speeding up the system, and adding some features to Trans. We applied the simple verifier to a directory-based cache-coherence protocol for use with a communication medium that does not preserve message order (for example, because of different routing for consecutive messages). Early in the design, we began working from a high-level summary of the protocol with several interesting findings:

1. Verification was worth the effort. There were several errors and many more oversights (issues that were not addressed in the design).

2. Almost all of the bugs in the system could be caught with very small configurations: for example, three processors with one memory location consisting of one bit. Hence, not many states needed to be explored (never over a million), so our simple verification algorithms were sufficient.

3. Verification by the designer early in the design, on a high-level model of the design, is very productive. In this mode, the formal verifier is much like a highly

critical design assistant who finds all the obscure cases the designer failed to think about.

4. Adding error statements to (supposedly) unused branches of IF and CASE statements catches more than half of the errors. Having an error statement in the description language makes the job of specification much easier.

The program was also used successfully to assist in the detailed design of a link-level communications protocol by another student during a summer job at Sun Microsystems.

The third major subproject was to design and implement a more sophisticated verifier for TRANS based on "symbolic" methods. A symbolic method uses an implicit instead of explicit representation of the state space of the system. The symbolic representation of the state space should grow in proportion to the "complexity" of the space, not just the number of states. One of the most widely-used symbolic representations is the "binary decision diagram", a directed graph representation of boolean functions.

Our method uses binary decision diagrams to represent a finite state graph representing the behavior of the protocol. The state space of the protocol can be explored using breadth-first search, which is performed efficiently by boolean manipulations. Although these techniques have been used by other groups, the novelty of our approach is that the boolean representation is derived automatically from the high-level protocol descriptions. Other systems have required the protocol description to be written in boolean form. We have implemented this verifier, which works on small examples. We are now in the process of debugging and performance tuning.

Liveness and Fairness

Murphi is a description language for concurrent systems (such as protocols) that we have been developing and using for several years. A Murphi description consists of a set of guarded commands (condition/action rules) which are iterated indefinitely. The Murphi verifier generates the states of a system by systematically applying the guarded commands in all possible ways, checking each state as it is generated against a set of user-specified properties. If one of the properties is violated, the Murphi verifier generates an example execution history of the system to show how the error can occur. Murphi has been used to debug several descriptions at universities and in industry, including the DASH cache coherence and FLASH cache coherence protocols. As part of this contract, a version of Murphi was documented and distributed to both industry and university researchers.

Verification of "Relaxed Memory Order"

An important application of the new Murphi liveness checker was a Murphi model of "Relaxed Memory Order" (RMO), which is a new standard memory model for the Sparc V9 architecture, defined by the Sparc International Consortium. RMO is a very flexible multiprocessor memory model that allows memory loads and stores to distinct locations to be reordered very freely.

We developed an operational description in Murphi of the RMO model. This model could be coupled in Murphi with small Sparc V9 assembly codes to verify the correctness of synchronization routines or to generate all the possible results of several interacting programs

under the memory model.

Our initial system did not allow us to verify that the synchronization functions satisfied liveness requirements. A mutual exclusion algorithm should have the property that if a process tries to enter a critical section it will not have to wait forever. In order to prove such properties with the new liveness checker, it is first necessary to specify certain "fairness properties" of the memory model. For example, if a store is issued by a processor, it should eventually be executed by the memory; also, although the processors may vary widely in speed, each processor eventually issues its next instruction.

We were able to verify all of our previous examples with the expected results (including some livelocks). One of the most interesting examples was one we had not previously verified: a version of Dekker's algorithm with a starvation condition. The problem occurs when two processors repeatedly try to enter the critical section in exactly the same way, so each is blocked. The verifier was able to detect the problem and print the correct diagnostic trace.

Integration with Symmetry Reduction

One of the most successful ideas we have used in Murphi is "symmetry reduction". Many problems have natural structural symmetries where interchanging values in a state simply interchanges values in all future states, without otherwise changing the computation that occurs. For example, in a multiprocessor cache coherence protocol such as that used in DASH, the processors, memory addresses, and data values can all be interchanged without really changing the state: if no error occurs when processor 1 has a clean copy of a cache line and processor 2 has a dirty copy, no error will occur when processors 1 and 2 are exchanged.

Two states can be regarded as equivalent if one can be obtained from the other by interchanging symmetric values in this way. Huge savings by skipping a state whenever an equivalent state has already been processed; two orders of magnitude for DASH and even more for other cache coherence protocols.

We have begun integrating symmetry reduction with the liveness algorithms. Although everyone believed that this would be straightforward, we have discovered that there are some surprising interactions and that the problem is not quite as simple as believed. Intuitively, problems arise because a liveness property may require that two particular states be visited on any execution. However, symmetry reduction may implicitly merge the two states, losing track of whether both were visited or only one. As of the close of this contract, we have not found an ideal solution to this problem.

FLASH Protocol Verification

We also set out to verify the cache coherence protocol for the FLASH multiprocessor. The protocol was described in Murphi. The resulting operational model is a finite state concurrent system composed of several processes modeling processors, caches, directory processors (DP), a pool of pointers, and networks connecting processors. The DP is the real part implementing the protocol and consists of local memory and directory for each memory location. States in the model consist of global variables which represent the states of

each processes. Processors and caches are modelled abstractly in that only the necessary information for the cache coherence protocol is maintained. On the other hand, detailed implementations in the DP such as information bits in directory header are modelled by Boolean variables in Murphi description as they are in the specification.

With the data structure, the protocol is described by transition relations among the states of the machine. The FLASH protocol specification fits with the rule-based nature of Murphi very well. The specification consists of a number of handlers for each of request and reply message type. A handler is specified by a table which simply lists case by case actions depending on the directory information of the memory line in the DP. Each of table entry can be directly translated into Murphi rules in a straightforward way.

To keep the size of the state space manageable for verification, we have to confine the number of resources of the system within small numbers. Thus, the largest system verified successfully is with four processors and a single memory location. However, within these limits, the Murphi verifier has explored EVERY POSSIBLE scenario, so it is likely that significant errors would have been caught by this process.

The specifications that were checked using the Murphi verifier included in-line assertions, invariants, liveness properties, and absence of deadlock.

Assertions:

Murphi allows the user to insert assertions and error statements in the middle of a description. For example, when the user writes an "if" statement and the "else" clause is never supposed to be executed, it is possible to insert an error statement in the else. If it is possible for the else clause to be executed, Murphi will catch it and issue an error message and diagnostic execution that leads to the problem case. This feature is used frequently in the FLASH description, especially for detecting the arrival of messages of unexpected types.

Deadlocks:

Running the verifier on the model with limited number of resources, we found deadlocks, especially when network queues were considered to be very small. In the actual implementation, FLASH uses a software queue whose size is virtually infinite. When we limited the number of transactions in the system to avoid overflowing the queues in the model, there were no further deadlocks.

Invariants:

Besides the assertions, two groups of invariants were added in Murphi description. One group specifies properties of the directory headers in the DP implementation and the other specifies general properties of cache coherence protocols. These invariants are expected to hold at every reachable state of the system.

Liveness Properties:

Murphi is able to check liveness properties under the weak fairness assumption. The properties are written in a subset of linear time temporal logic. This will capture livelocks of the protocol if they exist, and violations of the given temporal specifications. We have tested two kinds of temporal requirements. First, for each of the memory location, the directory

header has a pending bit which tells if there are any unacknowledged requests on that memory location. This bit should be eventually cleared, so that any request to this location are processed. Second, for each cache line in a processor, an information bit is assigned to decide if the line is expecting some replies to its requests. These requests should be satisfied eventually.

Verification Results:

Some of the properties are found to be violated during verification, but all of them were caused by trivial errors in the specification (there were no problems in the implementation). Evidently, either the designer violated the specification in exactly the right ways so that the implementation did the "right thing" when the specification did not, or the specification was not updated when problems were found during implementation. Hence, the advantages of the verification effort were to increase our confidence in the correctness of the protocol implementation, and to debug the specification.

Table 1 shows selected verification cases including the number of states explored and consumed time during verification.

Table 1:

# of Proc.	# of Memory locs.	Data values	Input queue	Network queue	Dup. pointer allowed	# of States	Time in seconds
2	2	yes	no	3	yes	205,586	12,660
3	1	no	yes	3	no	1,008,146	32,232
3	1	yes	no	3	no	1,021,464	39,756
4	1	no	no	2	no	256,081	11,176

The last line of the table is for a version of the FLASH description that was rewritten to exploit the fact that all processors, values, and cache lines are interchangeable at this level of abstraction. This allows the elimination of many redundant states, allowing larger descriptions to be verified.

Publications

1. Adve, S., Gharachorloo, K., Gupta, A., Hennessy, J., et al. "Sufficient System Requirements for Supporting the PLPC Memory Model". Stanford University, Computer Systems Laboratory. Technical Report, CSL-TR-93-595. December, 1993.

2. Altman, R.B., Chen, C.C., Poland, W.B., and Singh, J.P. "Probabilistic Constraint Satisfaction with Non-Gaussian Constraint Noise" in Tenth Annual Conference on Uncertainty in Artificial Intelligence. Seattle, WA. July 1994.

3. Amarasinghe, S., Anderson, J., Lam, M.S., and Lim, A. "An Overview of a Compiler for Scalable Parallel Machines" in 6th Workshop on Languages and Compilers for Parallel Computing. August 1993.

4. Amarasinghe, S.P., Anderson, J.M., Lam, M.S., and Tseng, C.-W. "Design and Evaluation of Compiler Optimizations for Scalable Shared Address Space Machines" in Architectural Support for Programming Languages and Operating Systems Conference (ASPLOS VI). San Jose, CA. October 1994.

5. Amarasinghe, S., Anderson, J., Lam, M.S., and Tseng, C.-W. "An Overview of the SUIF Compiler for Scalable Parallel Machines" in Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing. San Francisco, CA. February 1995.

6. Anderson, J. and Lam, M.S. "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines" in '93 Conference on Programming Language Design and Implementation. ACM SIGPLAN. June 1993.

7. Baskett, F. and Hennessy, J., "Microprocessors: From Desktops to Supercomputers." Science. Vol. 261: pgs. 864-871. August 1993.

8. Chanak, T., Horowitz, M., Lacroute, P., Maneatis, J., et al., Implications of Non-Binary-Sized Instructions. 1992

9. Chandra, R., Gupta, A., and Hennessy, J.L. "Integrating Concurrency and Data Abstraction in a Parallel Programming Language". Stanford University, Computer Systems Laboratory. Technical Report, CSL-TR-92-511. February, 1992.

10. Chandra, R., Gupta, A., and Hennessy, J. "Data Locality and Load Balancing in COOL" in 4th Symposium on the Principles and Practice of Parallel Programming. ACM SIGPLAN. San Diego, CA. pgs. 249-259. May 1993.

11. Chandra, R., Gupta, A., and Hennessy, J.L., "Integrating Concurrency and Data Abstraction in the COOL Programming Language." Computer. February 1994.

12. Chandra, R., Gharachorloo, K., Soundararajan, V., and Gupta, A. "Performance Evaluation of Hybrid Hardware and Software Distributed Shared Memory Protocols" in Proceedings of the 8th International Conference on Supercomputing. ACM. July 1994. Also appears as Stanford University technical report no. CSL-TR-93-597 published December 1993.

13. Chandra, R., Devine, S., Verghese, B., Gupta, A., et al. "Scheduling and Page Migration for Multiprocessor Compute Servers" in Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI). ACM/IEEE. San Jose, CA. pgs. 12-24. October 1994.

14. Chandra, R., Gupta, A., and Hennessy, J., "COOL: An Object-Based Language for Parallel Programming." IEEE Computer. Vol. 27(8): pgs. 14-26. August 1994.

15. Davis, H., Goldschmidt, S.R., and Hennessy, J. "Multiprocessor Simulation and Tracing Using Tango" in International Conference on Parallel Processing (ICPP). St. Charles, IL. August 1991.

16. Davis, H.M. "Multiprocessor Simulation: Achieving Accuracy, Efficiency, and Flexibility." Ph.D. Thesis from Stanford University, Computer Systems Laboratory, October 1993.

17. Dean, M.E., Dill, D.L., and Horowitz, M., "Self-Timed Logic Using Current-Sensing Completion Detection (CSCD)." Journal of VLSI Signal Processing. Vol. 6(3): 1993. Also in IEEE International Conference on Computer Design: VLSI In Computers and Processors, Cambridge, MA, pp. 187-191, Oct. 1991.

18. Dill, D., Hu, A.J., Drexler, A., and Yang, C.H. "Protocol Verification as a Hardware Design Aid" in International Conference on Computer Design. Cambridge, MA. October 1992.

19. Dill, D.L., Park, S., and Nowatzyk, A., Formal Specification of Abstract Memory Models, in Research on Integrated Systems: Proceedings of the 1993 SymposiumG. Borriello and Ebeling, C. 1993, MIT Press: Boston, MA. 38-52.

20. Erlichson, A.J., Nayfeh, B.A., Singh, J.P., Olukotun, K., et al. "The Benefits of Clustering in Shared Address Space Multiprocessors: An Applications-Driven Investigation" in Architectural Support for Programming Languages and Operating Systems Conference (ASPLOS VI). San Jose, CA. October 1994.

21. Gharachorloo, K., Gupta, A., and Hennessy, J. "Hiding Memory Latency using Dynamic Scheduling in Shared-Memory Multiprocessors" in 19th International Symposium on Computer Architecture. IEEE/ACM. Queensland, Australia. pgs. 22-33. May 1992. CSL-TR-93-567.

22. Gharachorloo, K., Adve, S., Gupta, A., Hennessy, J., et al., "Programming for Different Memory Consistency Models." Journal of Parallel and Distributed Computing. Vol. 15(4): pgs. 399-407. 1992.

23. Gharachorloo, K.G., Anoop and Hennessy, John. "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors (Revision to)". Stanford University, Computer Systems Laboratory. Technical Report, CSL-TR-93-568. April, 1993.

24. Gharachorloo, K., Adve, S., Gupta, A., Hennessy, J., et al. "Specifying System Requirements for Memory Consistency Models". Stanford University, Computer Systems Laboratory. Technical Report, CSL-TR-93-594. December, 1993.

25. Goldberg, A.J. "Reducing Overhead in Counter-Based Execution Profiling". Stanford University, Computer Systems Laboratory. Technical Report, CSL-TR-91-495. October 1991.

26. Goldberg, A.J. and Hennessy, J.L., "MTOOL: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications." Transactions on Parallel and Distributed Systems. Vol. 4(1): pgs. 28-40. 1993. Special issue on Measurement & Analysis.

27. Goldschmidt, S.R. and Hennessy, J.L. "The Accuracy of Trace-Driven Simulations of Multiprocessors" in SIGMETRICS Conference on Measurement and Modeling of Computer Systems. ACM. Santa Clara, CA. May 1993.

28. Gupta, A., Hennessy, J., Gharachorloo, K., Mowry, T., et al. "Comparative Evaluation of Latency Reducing and Tolerating Techniques" in 18th International Symposium on Computer Architecture (ISCA). IEEE/ACM. Toronto, Canada. pgs. 254-263. May 1991. Also appears as Stanford Univ. Technical Report No. CSL-TR-91-467, March 1991.

29. Gupta, A., Joe, T., and Stenstrom, P. "Comparative Performance Evaluation of Cache-Coherent Numa and Coma Architectures". Stanford University, Computer Systems Laboratory. Technical Report, CSL-TR-92-524. May, 1992.

30. Hall, M.W., Mellor-Crummey, J.M., Carle, A., and Rodriguez, R.G. "FIAT: A Framework for Interprocedural Analysis and Transformation" in 6th Annual Languages and Compilers for Parallel Computing Workshop. Portland, OR. August 1993. Also published in "Lecture Notes in Computer Science" by Springer-Verlag, 1994.

31. Hall, M.W., Murphy, B., and Amarasinghe, S. "Interprocedural Parallelization Analysis: A Case Study" in Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing. San Francisco, CA. February 1995.

32. Heinlein, J., Gharachorloo, K., Dresser, S., and Gupta, A. "Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor" in Architectural Support for Programming Languages and Operating Systems Conference (ASPLOS VI). San Jose, CA. pgs. 38-50. October 1994.

33. Heinlein, J., Gharachorloo, K., and Gupta, A. "Integrating Multiple Communication Paradigms in High Performance Multiprocessors". Stanford University, Computer Systems Laboratory. Technical Report, CSL-TR-94-604. February, 1994.

34. Heinrich, M., Kuskin, J., Ofelt, D., Heinlein, J., et al. "The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor" in Architectural Support for Programming Languages and Operating Systems Conference (ASPLOS VI). IEEE/ACM. San Jose, CA. pgs. 274-285. October 1994.

35. Hennessy, J.L. and Patterson, D.A., Computer Organization and Design: The Hardware/Software Interface. First ed. 1993, San Mateo, CA: Morgan Kaufmann Publishers.

36. Hennessy, J.L., Architectural Convergence and Its Implications, in Developing a Computer Science Agenda for High-Performance Computing Ed. U. Vishkin. 1994, Association for Computing Machinery: New York. 72-78.

37. Herrod, S., Witchel, E., Rosenblum, M., and Gupta, A., "Fast and Accurate Multiprocessor Simulation: The SimOS Approach." IEEE Computer (Special Issue). September 1995.

38. Horowitz, M. and Keutzer, K. "Hardware-Software Co-Design" in Synthesis and Simulation Meeting and International Interchange (SASIMI) '93. Nara, Japan. October 1993. Invited paper.

39. Hu, A.J., Dill, D., Drexler, A., and Yang, C.H. "Higher-Level Specification and Verification with BDDs" in Proceedings of the Workshop on Computer-Aided Verification. Quebec, Ontario, Canada. July 1992.

40. Hu, A.J. and Dill, D. "Reducing BDD Size by Exploiting Functional Dependencies" in Design Automation Conference. Dallas, TX. June 1993.

41. Joe, T. and Hennessy, J.L. "Evaluating the Memory Overhead Required for COMA Architectures" in 21st Annual International Symposium on Computer Architecture (ISCA). IEEE/ACM. Chicago, IL. pgs. 82-93. April 1994.

42. Kao, R. and Horowitz, M. "Asymptotic Waveform Evaluation for Circuits With Redundant DC Equations". Stanford University, Computer Systems Laboratory. Technical Report, CSL-TR-91-478. May 1991.

43. Kao, R. and Horowitz, M. "Efficient Moment-Based Timing Analysis for Variable Accuracy Switch Level Simulation". Stanford University, Computer Systems Laboratory. Technical Report, CSL-TR-91-468. April 1991.

44. Kao, R. and Horowitz, M. "Piecewise Linear Models for Rsim" in International Conference on Computer Aided Design (ICCAD). Santa Clara, CA. November 1993.

45. Kao, R. and Horowitz, M., "Eliminating Redundant DC Equations for Asymptotic Waveform Evaluation." Transactions on Computer-Aided Design. Vol. 13(3): pgs. 396-397. March 1994.

46. Kozlov, A.V. and Singh, J.P. "A Parallel Lauritzen-Spiegelhalter Algorithm for Probabilistic Inference" in Tenth Annual Conference on Uncertainty in Artificial Intelligence. Seattle, WA. July 1994.

47. Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., et al. "The Stanford FLASH Multiprocessor" in 21st Annual International Symposium on Computer Architecture (ISCA). IEEE/ACM. Chicago, IL. pgs. 302-313. April 1994.

48. Lam, M.S. and Wolf, M.E. "Automatic Blocking by a Compiler" in Fifth SIAM Conference on Parallel Processing for Scientific Computing. March 1991.

49. Lam, M.S. and Wilson, R.P. "Limits of Control Flow on Parallelism" in 19th International Symposium on Computer Architecture (ISCA). IEEE/ACM. Queensland, Australia. pgs. 46-57. May 1992.

50. Lam, M. "Locality Optimizations for Parallel Machines" in Third Joint International Conference on Vector and Parallel Processing. CONPAR. Linz, Austria. pgs. 17-28. September 1994. Keynote Address.

51. Laudon, J., Gupta, A., and Horowitz, M. "Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations" in Architectural Support for Programming Languages and Operating Systems Conference (ASPLOS VI). San Jose, CA. pgs. 308-318. October 1994.

52. Laudon, J., Gupta, A., and Horowitz, M., Architectural and Implementation Tradeoffs in the Design of Multiple-Context Processors, in Multithreaded Computer Architectures. Kluwer: 1994.

53. Lenoski, D., Laudon, J., Joe, T., Nakahira, D., et al. "The DASH Prototype: Implementation and Performance" in 19th International Symposium on Computer Architecture. IEEE/ACM. Queensland, Australia. pgs. 92-103. May 1992.

54. Lenoski, D., Laudon, J., Joe, T., Nakahira, D., et al., "The DASH Prototype: Logic Overhead and Performance." Transactions on Parallel and Distributed Systems. Vol. 4(1): pgs. 41-61. January 1992.

55. Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., et al., "The Stanford DASH Multiprocessor." Computer. Vol. 25(3): pgs. 63-79. 1992.

56. Lenoski, D.E. "The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor." Ph.D. Thesis from Stanford University, Computer Systems Laboratory, February 1992.

57. Lim, A.W. and Lam, M.S. "Communication-Free Parallelization via Affine Transformations" in Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing. August 1994.

58. Maneatis, J. and Horowitz, M., "Precise Delay Generation Using Coupled Oscillators." IEEE Journal of Solid-State Circuits. Vol. 28(12): pgs. 1273-1282. December 1993. A shorter version appeared in Proceedings of the ISSCC in San Francisco, CA, pp.118-119, February, 1993.

59. Martonosi, M., Gupta, A., and Anderson, T. "MemSpy: Analyzing Memory System Bottlenecks in Programs" in Conference on the Measurement and Modeling of Computer Systems. ACM SIGMETRICS. pgs. 1-12. June 1992.

60. Martonosi, M., Gupta, A., and Anderson, T. "Effectiveness of Trace Sampling for Performance Debugging Tools" in SIGMETRICS. Santa Clara, CA. May 1993.

61. Martonosi, M., Gupta, A., and Anderson, T., "Tuning Memory Performance in Sequential and Parallel Programs." IEEE Computer. 1994.

62. Maydan, D.E., Hennessy, J.L., and Lam, M.S. "Efficient and Exact Data Dependence Analysis" in Conference on Programming Language Design and Implementation (PLDI). ACM SIGPLAN. Toronto, Ontario, Canada. June 1991.

63. Maydan, D. "Accurate Analysis of Array References." Ph.D. Thesis from Stanford

University, Computer Systems Laboratory, 1992.

64. Maydan, D., Amarasinghe, S., and Lam, M.S. "Array Data Flow Analysis and its Use in Array Privatization" in 20th Annual Symposium on Principles of Programming Languages. ACM. pgs. 1-14. January 1993.

65. McFarling, S. "Program Analysis and Optimization for Machines with Instruction Cache." Ph.D. Thesis from Stanford University, Computer Systems Laboratory, September 1991.

66. Mowry, T. and Gupta, A., "Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors." Journal of Parallel and Distributed Computing. Vol. 12(2): pgs. 87-106. June 1991.

67. Mowry, T., Lam, M.S., and Gupta, A. "Design and Evaluation of a Compiler Algorithm for Prefetching" in Proceedings of the 5th International Conference on Architectural Support for Programming Languages. IEEE/ACM. Boston, MA. pgs. 248-261. October 1992.

68. Nayfeh, B. and Olukotun, K. "Exploring the Design Space for a Shared-Cache Multiprocessor" in Intl. Symposium on Computer Architecture. ACM/IEEE. Chicago, IL. April 1994.

69. Nieh, J. and Levoy, M. "Volume Rendering on Scalable Shared-Memory MIMD Architectures" in Boston Workshop on Volume Visualization. Boston, MA. October 1992.

70. Nowick, S.M., Dean, M.E., Dill, D.L., and Horowitz, M. "The Design of a High-Performance Cache Controller: A Case Study in Asynchronous Synthesis" in 26th Annual Hawaii International Conference on System Sciences. Maui, HI. pgs. 418-427. Jan. 1993. A modified version appears in a special issue of Integration: The VLSI Journal, Vol. 115, pp. 241-262, 1993, Elsevier Science Publishers B.V.

71. Ohara, M. and Hennessy, J. "Deliver: Combining the Invalidation and Update Protocol" in 21st Annual International Symposium on Computer Architecture (ISCA). IEEE/ACM. Chicago, IL. April 1994.

72. Ohara, M. and Hennessy, J. "Consumer-based versus Producer-based Prefetch" in Architectural Support for Programming Languages and Operating Systems Conference (ASPLOS VI). San Jose, CA. October 1994.

73. Pieper, K.L. "Parallelizing Compilers: Implementation and Effectiveness." Ph.D. Thesis from Stanford University, Computer Systems Laboratory, 1993.

74. Rinard, M.C. and Lam, M.S. "Semantic Foundations of Jade" in 19th Annual Symposium on Principles of Programming Languages. ACM. Albuquerque, NM. pgs. 105-118. January 1992.

75. Rinard, M.C., Scales, D.J., and Lam, M.S., "Jade: A High-Level, Machine-Independent Language for Parallel Programming." Computer. Vol. 26(6): pgs. 28-38. June 1993.

76. Rosenblum, M. and Varadarajan, M. "Fast Operating System Simulation" in Architectural Support for Programming Languages and Operating Systems Conference (ASPLOS VI). San Jose, CA. October 1994.

77. Rothberg, E. and Gupta, A. "The Performance Impact of Data Reuse in Parallel Dense Cholesky Factorization". Stanford University, Computer Systems Laboratory. Technical Report, CSL-54-92-503/STAN-CS-92-1401. January, 1992.

78. Rothberg, E. and Gupta, A., "An Evaluation of Left-Looking, Right-Looking and Multifrontal Approaches to to Sparse Cholesky Factorization on Hierarchical-Memory Machines." Journal on High Speed Computing. 1992. Also appears as Stanford technical report STAN-CS-91-1377/CSL-TR-91-487.

79. Rothberg, E. and Gupta, A. "An Efficient Block-Oriented Approach to Sparse Cholesky Factorization". Stanford University, Computer Systems Laboratory. Technical Report, STAN-CSL-92-533. July, 1992.

80. Rothberg, E., Gupta, A., Ng, E., and Peyton, B. "Parallel Sparse Matrix Factorization Algorithms for Shared-Memory Multiprocessor Systems" in Proceedings of the 7th International Conference on Computer Methods for Partial Differential Equations. 1992.

81. Rothberg, E., Singh, J.P., and Gupta, A. "Working Sets, Cache Sizes and Node Granularity Issues for Large-Scale Multiprocessors" in 20th Annual International Symposium on Computer Architecture. IEEE/ACM. San Diego, CA. pgs. 14-25. May 1993.

82 Scales, D., Rinard, M., Lam, M., and Anderson, J., "Hierarchical Concurrency in Jade." Languages and Compilers for Parallel Computing: Springer Verlag. pgs. 50-64. January 1992.

83. Schnorf, P., Ganapathi, M., and Hennessy, J., "Compile-time Copy Elimination." Software - Practice and Experience. Vol. 23(11): pgs. 1175-1200. November 1993.

84. Sidiropoulos, S., Yang, C.-K.K., and Horowitz, M. "A CMOS 500 Mbps/pin Synchronous Point to Point Link Interface" in 1994 Symposium on VLSI Circuits. Honolulu, HI. June 1994.

85. Simoni, R. and Horowitz, M. "Modeling the Performance of Limited Pointers Directories for Cache Coherence" in 18th International Symposium on Computer Architecture (ISCA),. Toronto, Ontario, Canada. pgs. 309-318,. May 1991.

86. Simoni, R. and Horowitz, M. "Dynamic Pointer Allocation for Scalable Cache Coherence Directories" in International Symposium on Shared Memory Multiprocessing. Tokyo, Japan. pgs. 72-81. April 1991. Also appears as Stanford University Technical Report, CSL-TR-91-491, Aug. 1991.

87. Simoni, R. "Cache Coherence Directories for Scalable Multiprocessors". Stanford University, Computer Systems Laboratory. Ph.D. Thesis Report, July, 1992.

88. Singh, J.P., Hennessy, J.L., and Gupta, A. "Implications of Hierarchical N-body

Techniques for Multiprocessor Architecture" in 19th Annual International Symposium on Computer Architecture. Queensland, Australia. May 1992. Also appears as Stanford Technical Report CSL-TR-92-506.

89. Singh, J., Holt, C., Gupta, A., and Hennessy, J. "A Parallel Adaptive Fast Multipole Method" in Society for Industrial and Applied Mathematics - 40th Anniversary Conference. SIAM. Los Angeles, CA. July 1992.

90. Singh, J.P., Weber, W.-D., and Gupta, A., "SPLASH: Stanford Parallel Applications for Shared-Memory." Computer Architecture News. Vol. 20(1): March 1992. Also appears as Stanford Technical Report CSL-TR-91-469.

91. Singh, J.P. and Hennessy, J.L., "Finding and Exploiting Parallelism in an Ocean Simulation Program: Experiences Results, Implications." Journal of Parallel and Distributed Computing. Vol. 15(1): pgs. 27-48. May 1992.

92. Singh, J.P. "Parallel Hierarchical N-Body Methods and Their Implications for Multiprocessors". Stanford University, Computer Systems Laboratory. Ph.D. Thesis Report, CSL-TR-93-565. February, 1993.

93. Singh, J.P., Hennessy, J., and Gupta, A., "Scaling Parallel Programs for Multiprocessors: Methodology and Examples." Computer. Vol. 26(7): pgs. 42-50. July 1993. Also appears as Stanford technical report CSL-TR-92-541.

94. Singh, J., Joe, T., Gupta, A., and Hennessy, J.L. "An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessors" in Supercomputing '93. Portland, OR. November 1993.

95. Singh, J.P., Holt, C., Hennessy, J.L., and Gupta, A. "A Parallel Adaptive Fast Multipole Method" in Supercomputing '93. Portland, OR. November 1993.

96. Singh, J.P., Gupta, A., and Levoy, M., "Parallel Visualization Algorithms and their Implications for Multiprocessor Architecture." Computer (Special Issue on Visualization). Vol. 27(7): pgs. 45-56. July 1994.

97. Singh, J.P., Holt, C., Totsuka, T., Gupta, A., et al., "Load Balancing and Data Locality in Hierarchical N-Body Methods." Journal of Parallel and Distributed Computing. 1994. Also appears as Stanford Technical Report CSL-TR-92-505.

98. Singh, J.P., Rothberg, E., and Gupta, A. "Modeling Communication in Parallel Algorithms: A Fruitful Interaction between Theory and Systems?" in Sixth Annual Symposium on Parallel Algorithms and Architectures. ACM. Cape May, NJ. June 1994.

99. Smith, M.D. "Tracing with Pixie". Stanford University, Computer Systems Laboratory. Technical Report, CSL-TR-91-497. November, 1991.

100. Smith, M.D., Horowitz, M., and Lam, M.S. "Efficient Superscalar Performance Through Boosting" in 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). IEEE/ACM. Boston, MA. pgs. 248-259. Oct. 1992.

101. Smith, M. "Support for Speculative Execution in High-Performance Processors." Ph.D. Thesis from Stanford University, Computer Systems Laboratory, 1992.

102. Soule, L. and Gupta, A., "An Evaluation of the Chandy-Misra-Bryant Algorithm for Digital Logic Simulation." ACM Transactions on Modeling and Computer Simulation Vol. 1(4): October 1992.

103. Soule, L. and Gupta, A. "Evaluation of the Chandy-Misra-Bryant Algorithm for Digital Logic Simulation (An)" in 1992 SCS Western Simulation Multiconference. Newport Beach, CA. January 1992.

104. Soule, L.P. "Parallel Logic Simulation: An Evaluation of Centralized-Time and Distributed-Time Algorithms." Ph.D. Thesis from Stanford University, Computer Systems Laboratory, June 1992.

105. Stenstrom, P., Joe, T., and Gupta, A. "Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures" in International Symposium on Computer Architecture. IEEE/ACM. Queensland, Australia. May 1992.

106. Tjiang, S., Wolf, M., Lam, M., Pieper, K., et al., Integrating Scalar Optimization and Parallelization, in Languages and Compilers for Parallel Computing, Eds. H. Goos, G. Banerjee, Nicolau, Padua. 1992, Springer-Verlag: New York. 137-151.

107. Tjiang, S.W.K. and Hennessy, J. "Sharlit--A Tool for Building Optimizers" in Proceedings of on Programming Language Design and Implementation. ACM SIGPLAN. pgs. 82-93. June 1992.

108. Tjiang, S.W.K. "Automatic Generation of Data Flow Analyzers: A Tool for Building Optimizers." Ph.D. Thesis Thesis from Stanford University, Computer Systems Laboratory, 1993.

109. Torrellas, J., Gupta, A., and Hennessy, J. "Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System" in Conference on Architectural Support for Programming Languages and Operating Systems. IEEE/ ACM. Boston, MA. pgs. 162-174. October 1992. Also appears as Stanford technical report CSL-TR-92-512.

110. Torrellas, J., Tucker, A., and Gupta, A. "Evaluating the Benefits of Cache-Affinity Scheduling in Shared-Memory Multiprocessors". Stanford University. Technical Report, CSL-TR-92-536. August, 1992.

111. Torrellas, J., Gupta, A., and Hennessy, J. "Characterizing the Cache Performance and Synchronization Behavior of a Multiprocessor Operating System". Stanford University, Computer Systems Laboratory. Technical Report, CSL-TR-92-512. January, 1992.

112. Torrellas, J., Lam, M.S., and Hennessy, J.L., "False Sharing and Spatial Locality in Multiprocessor Caches." Transactions on Computers. 1994.

113. Williams, T.E. and Horowitz, M.A. "A 160nS 54bit CMOS Division Implementation using Self-Timing and Symmetrically Overlapped SRT Stages" in IEEE ARITH 10

Conference. Grenoble, France. pgs. 210-217. June 1991.

114. Williams, T.E. and Horowitz, M.A., "A Zero-Overhead Self-Timed 160-ns 54-b CMOS Divider." IEEE Journal of Solid-State Circuits. Vol. 26(11): pgs. 1651-1661. November 1991. Shorter version in Proceedings of International Solid-State Circuits Conference (ISSCC), San Francisco, CA, pp. 98-99, February 1991.

115. Wilson, R., French, R., Wilson, C., Amarasinghe, S., et al. "SUIF: A Parallelizing and Optimizing Research Compiler". Stanford University - Computer Systems Laboratory. Technical Report, CSL-TR-94-620. May 1994.

116. Wilson, R., French, R., Wilson, C., Amarasinghe, S., et al., "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers." ACM SIGPLAN Notices. 1994.

117. Wingard, D.E., Stark, D.C., and Horowitz, M.A., "Circuit Techniques for Large CSEA SRAMs." IEEE Journal of Solid-State Circuits. Vol. 27(6): pgs. 908-919. June 1992.

118. Wolf, M. "Improving Locality and Parallelism in Nested Loops." Ph.D. Thesis from Stanford University, Computer Systems Laboratory, 1992.

119. Woo, S., Singh, J.P., and Hennessy, J.L. "The Performance Advantages of Integrating Message-Passing in Cache-Coherent Multiprocessors". Stanford University, Computer Systems Laboratory. Technical Report, CSL-TR-93-593. November, 1994.

120. Woo, S.C., Singh, J.P., and Hennessy, J.L. "The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors" in Architectural Support for Programming Languages and Operating Systems Conference (ASPLOS VI). IEEE/ACM. San Jose, CA. pgs. 219-231. October 1994.